Scalable Principal Component Analysis On Husky

BAO Ergute

Dept. of Computer Science and Engineering
the Chinese University of Hong Kong

Abstract

Principal Component Analysis is a powerful tool for retrieving information and gain valuable insights from large matrices(data). During this intern, we implemented Scalable Principal Component Analysis(SPCA) that especially fit for big data on a distributed platform called Husky, implemented by Prof. Cheng and his PhDs. Then we compared the performance of our SPCA implementation with that of the basic Principal Component Analysis of MLlib on Spark. The result shows that SPCA on Husky outperforms MLlib on Spark a lot when the input data is extremely large.

Introduction

- **Big Data** Big data is a term for data sets that are so large or complex that traditional data processing applications are inadequate. Challenges include analysis, capture, data curation, search, storage, transfer, etc. The main challenge for SPCA implementation is analysis and storage.
- **Husky's Architecture** Husky Husky is a distributed computing system. Husky's basic architecture consists of a master representing an application, a coordinator managing the workers inside this process and several workers representing how CPU cores work with their associated data.
- **Husky C++ Basics and APIs** In Husky, the basic unit in Husky is an object, which is like the *object* in Object-Orientation of C++. Husky provides programmers with some useful APIs, such as **Broadcast**, **Send_Message** and **Request**. Moreover, **List_Execute** can iterate through objects in a list and achieve synchronization in the meantime. In our implementation, we mainly used **Aggregator**, which aggregates the results during a **List_Execute**.
- **Principal Component Analysis** Principal component analysis (PCA) is a statistical procedure that convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

Pseudo Code

input : $Y \in R^{N \times D}$
output: $C \in R^{D \times d}$
$C = normrnd(D, d)$;
$ss = normrnd(1, 1)$;
$Ym = mean(Y)$ **parallel**;
$ss1 = FrobeniusNorm(Y)$ **parallel**;
while *not* $STOP_CONDITION$ **do**
  $M = C^T * C + ss * I$;
  $CM = C * M^{-1}$;
  $Xm = Ym * CM$;
  $XtX, YtX =$ ***YtXJob***(Y, Ym, Xm, CM) ***parallel***;
  $XtX += ss * M^{-1}$;
  $C = YtX / XtX$;
  $ss2 = trace(XtX * C^T * C)$;
  $ss3 =$ ***ss3Job***(Y, Ym, Xm, CM, C) ***parallel***;
  $ss = (ss1 + ss2 - 2 * ss3)$;
end

Details Of SPCA

We adopted the algorithm and pseudo code of SPCA[1] from Elgamal's paper and we also did some refinement. SPCA is a scalable version of Probabilist Principal Component Analysis(PPCA). In this probabilistic approach, PCA is presented as a latent(unobserved) variable model that seeks a linear relation between a D-dimensional observed data vector y and a d-dimensional latent variable x. The model is defined by: $y = C * x + \mu + e$, where C is a $D \times d$ transformation matrix (i.e, the columns of C are the principal components), $x \sim N(0, I)$, μ is the vector mean of y, and $e \sim N(0, ss * I)$ is white noise to compensate for errors.[1] The work in [2] shows that, given N observations yrN1 as the input data, the log likelihood of data is given by:$\mathcal{L}(y_r) = \sum lnp(y_r)$.Thus, the *Maximum Likelihood Estimate* (MLE) of C is obtained by optimizing:

$$argmax_C \mathcal{L}(y_r) \quad (1)$$

The main idea behind the Probablistic PCA algorithm described in [2] is that the MLE solution of Equation [3] is equivalent to the solution of PCA. Moreover, [2] proposed an Expectation Maximization (EM) [4] algorithm to optimize the likelihood of Equation (1). EM is a well-known method to optimize the likelihood of models when a closed form solution does not exist. In our implementation, we obtain C by solving the maximum loglikelihood expectation iteratively, as they did in PPCA.

Implementation Details

There are four parts that involves parallel computation in the algorithm. The first two are done only once. The latter two are done in every iteration. In order to speed up the algorithm, our implementation focus on the code inside the while loop. In **YtXJob**, we do the following computation in parallel: $XtX = X^T * X + ss * M^{-1}$, $YtX = Yc^T * X$, where $X = Yc * C * M^{-1}$ and $Yc = Y - Ym$. First, we compute $X = y * C * M^{-1} - Ym * C * M^{-1}$ instead of computing $X = Yc * C * M^{-1}$ to leverage the sparsity in Y. Then we make use of the intermediate matrix X to compute XtX and YtX. Then we compute $YtX = Y^T * X - Ym^T * X$ instead of computing $YtX = Yc^T * X$ directly to leverage the sparsity in Y. In **ss3Job**, we compute $ss3 = \sum_{n=1}^{N} X_n * C^T * Yc_n^T$. Similarly, to leverage sparsity, we do $X_n * (C^T * Y_n^T) - X_n * C^T * Y_m^T$ since $C^T * Y_n^T$ is small and sparse. For the computation not done in parallel, we just do it in every single worker. Although it keeps the worker working intensively, it actually saves the time of communication between workers. In particular, our implementation of SPCA is tailored for large and sparse matrices.

Performance and Conclusion

Input	d	SPCA	MLlib
Matrix1	300	80sec	225sec
Matrix1	100	22sec	170sec
Matrix1	10	6sec	120sec
Matrix2	50	315sec	fail
Matrix3	10	100sec/iteration	fail

Matrix1 is of size $353 \times 2K$, i.e. $N = 353$ and $D = 2K$, Matrix2 is of size 64×47236 and Matrix3 is of size 20242×47236. SPCA was implemented and run on Husky. MLlib-PCA was run on Spark. All tests were run under the same environment. The $353 \times 2K$ input matrix is dense, the other 2 input matrices are sparse. As we can see, although Matrix1 is dense, our SPCA still outperforms PCA of MLlib. Moreover, when the input matrix is extremely fat(D big), MLlib can no longer do the computation.

Known Problems

The limitation of our SPCA implementation lies within with D, d and the memory limit of each worker. As we can see, each worker should be able to store matrices of size $D \times D$ and do matrix operations of the same size. Moreover, the speed is also limited by D and d, the smaller D and d are, the faster the computation is. In the future, we may need determine if the input matrix is sparse or dense during run time and come up with the optimal storing scheme. We may also need to look into **Eigen**, the library we called to do single-machine matrix operation, to see if there is anything to refine.

Conclusion

In conclusion, our SPCA is way better than MLlib when the input matrix is sparse, large and fat. Otherwise, it may still outperform MLlib. Last but not the least, MLlib is more suitable when the input matrix is tall and thin.

References

[1] T. Elgamal, M. Yabandeh, A. Aboulnaga, W. Mustafa, and M. Hefeeda. spca: Scalable principal component analysis for big data on distributed platforms. In SIGMOD, pages 79–91, 2015.

[2] M. E. Tipping and C. M. Bishop. Mixtures of probabilistic principal component analysers. Neural Computation, 11(2):443–482, 1999.

[3] Mahout machine learning library: http://mahout.apache.org/.

[4] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. PVLDB, 9(5):420–431, 2016.

Acknowledgement

The author would like to thank Professor James CHENG, his PhD candidates, LIU Jie and all other people working in the lab.